John von Neumann Institute for Computing

Implementing Critical Sections with the Shared Explicit Cache System in the Shared Memory Parallel Architectures

T. Madajczak, H. Krawczyk

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 699-706, 2006.

http://www.fz-juelich.de/nic-series/volume33

Implementing Critical Sections with the Shared Explicit Cache System in the Shared Memory Parallel Architectures

Tomasz Madajczak[a b], Henryk Krawczyk[a]

[a]Gdansk University of Technology, Faculty of Electronic, Telecommunication and Informatics, Dep. of Computer System Architectures, ul. Narutowicza 11/12, 80-233 Gdansk, Poland

[b]Intel Technology Poland Ltd. ul. Slowackiego 173, 80-247 Gdansk, Poland

1. Introduction

This document presents the new method for implementing critical sections in the shared memory parallel architectures such as multithreaded multiprocessors integrated on a die. The method bases on the Shared Explicit Cache System (SHECS) implemented in the multiprocessor. The method is derived from the Folding method [1][2] and is also similar to the cache-based synchronization technique proposed in [3][4]. The new method in comparison with the state-of-the-art methods eliminates their limitations and disadvantages such as scalability, starving, while it adds usage flexibility and copes with the hardware multithreading extensions. The document presents the system architecture equipped with the SHECS and also the algorithm to implement operating system or application level locking service.

2. Background

2.1. Folding method in network processors

Efficient use of shared resources is always connected with the need of a critical section implementation. Multicore and multiprocessors systems have the ability to rely on specific hardware support and capabilities to synchronize the particular processor cores within the die or the integrated platform. Hardware techniques for critical sections are available in the network processors and they are very efficient, but not always universal [1]. Software techniques are still available for the SMP systems and for cases when the hardware support isn't flexible. Therefore, there is a need for an efficient and flexible method that would address the new perspectives for the shared memory parallel architectures such as multithreaded multiprocessors systems. This document presents such a new method based on the use of SHECS. It is derived from the Folding method.

The Folding method was introduced in the Intel's network processors IXP 2000 [5] as a universal method for programming software critical section for the threads of a single microengine (that is a RISC processor). The method uses the microengine's internal local memory and CAM (content addressable memory) lookup engine that comprises an internal explicit cache system managed with software.

Folding caches the read data to be modified. It manages with the following critical section scenario that firstly reads a resource, then modifies it, and finally writes back the modified data. The read resource is stored within this system and considered locked if its use counter is greater than 0. Folding may occasionally lost the order of the threads entering the critical section, because in case of finding locked entry the algorithm repeats the section entering. This order lost means that the critical section may be starving for some threads, as they have no luck and they always repeat entering. Extending the Folding method onto a number of processors is not an easy task and also starving critical sections aren't useful in the general purpose parallel systems.

All these issues are solved in the SHECS-based method. It bases on the new explicit cache memory system architecture that eliminates the Folding's limitations and disadvantages. Thus, the SHECS-based method is more universal, flexible, and may have more applications.

2.2. Cache coherency

The parallel shared memory architectures built with using general purpose processors with internal caches may have implemented a mechanism for enforcing the coherence of internal caches. Hardware solutions for this problem are presented in [6][7], while software algorithms are discussed in [8][9]. Generally there are two approaches: implicit methods that hide the problem for the system user or software, and explicit methods assuming that the system user or software is aware of the problem, treats cache as a normal shared resource and solves it with cache-locking [4] or other locking method. The introduced method addresses the problem in the similar way as explicit methods. It assumes explicit cache-locking with integrated data transfers of the most recent cached data value and additionally it copes with hardware threading.

3. Shared Explicit Cache System

3.1. Architecture assumptions

Shared explicit cache system (SHECS)[1] consists of CAM banks controller, a number of CAM banks and some shared fast SRAM memory for caching. Figure 1 shows that the shared explicit cache system should be connected in a similar way as shared memory to the parallel processors P1-PN. The key assumption is providing a number of CAM banks that can work together or separately.

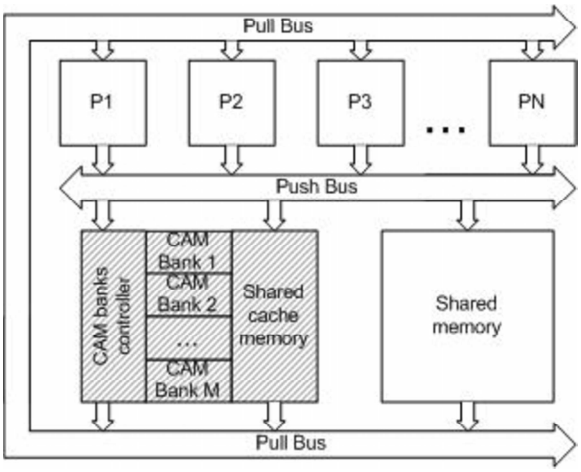

Figure 1. The SHECS in a shared memory parallel architecture

Every CAM bank is a functionally complete unit comprising entries tag and lock latches, lookup-locks FIFO queue and configuration for associated region of the cache memory. Every operation on the explicit cache system should have associated the mask that determines which banks are associated to the operation (single bit in the mask controls (enables/disables) one CAM bank). From the high level point of view the masking capability allows coupling a number of independent CAM

[1]The concept of the shared explicit cache system for network processors is patent pending in the U.S. patent office.

banks into a single CAM. Such a single CAM should also have a consistent old tag removing policy - least recently used tag should be removed upon adding a new tag within all the banks enabled with a particular mask. This requirement is realized with the CAM banks controller logic. The method of enforcing coherency of critical sections is hardware signaling. The signal arrival wakes up a virtual or hardware thread that normally waits on that signal for the completion of CAM-lookup-lock operation.

3.2. Functionality

A functionally complete SHECS should provide at least the following operations within the specified banks:

- CAM lookup with data locking and integrated reading
- CAM entry unlocking
- CAM entry's cache reading and writing with entry unlocking
- CAM banks managing (clearing the bank, adding, deleting tag, etc.)

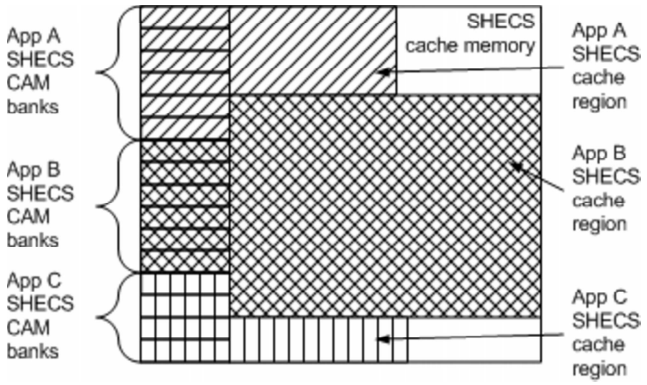

Figure 2. Example of the SHECS allocation among three parallel applications

The key feature of SHECS is ability to flexible divide onto independent subunits. This feature allows implementing a system wide SHECS allocation manager that on request allocates the specified amount of CAM entries. This capability is illustrated in Figure 2.

Every CAM bank may have associated a line of cache memory to store data to be locked and modified. Because, the memory is a part of the explicit cache memory system, the reading of such a memory line may be integrated with a successful CAM lookup operation. If processors P1-PN have a support for reading data bursts directly into the registers (such as network processors) this data may be used at once. Another way to use CAM lookup with integrated data transportation is to store data bursts within the processors' local caches. In that case the local cache may use the tag with the local processor address of this particular shared cache memory region or calculate the tag from the CAM entry index. The SHECS may handle some number of pending simultaneous CAM-lookup-locks. This limitation is connected with the depth L of the FIFO queue storing pending lookup-locks. Such a FIFO is implemented in the each CAM bank. The depth L may be exceeded

if a parallel application performs more than L simultaneous CAM-lookup-locks for the same tag value. In that case the CAM-lookup-lock requests are rejected with the result indicating locking fullness. In the other words the locking is not starving as long as the number of requesting task is lower and equal to the depth L. This depth should be chosen for a parallel system with considering the following factors:

- The number of N of parallel processors
- The number of K of hardware threads in each processor

The value of L should be calculated with the following formula:

$$L = N * K * C \tag{1}$$

where: N - the number of processors; K - the number of hardware threads in each processor; C - some constant $>= 1$

The value of depth L implemented according to formula (1) manages with all synchronization problems that use maximally all available parallel resources. However if a program tries to be executed with higher level of parallelism than available parallel resources (in the other words the program have more virtual threads or processes than there is hardware threads in the parallel system), then the locking implemented with using the explicit cache system may be starving for some virtual threads.

4. Implementing SHECS-based Full-Locking Critical Sections

Full-locking technique (described well in [10]) assumes that every element that must be used or modified coherently may be locked independently for a certain time. Because the explicit cache system has hardware limitations such as limited number of CAM entries and limited number of pending locks for a particular CAM bank (due to the limited depth L), the implementation of full-locking critical sections should combine the explicit cache system functionality and some OS (operating system) mechanismism to queue rejected SHECS requests. Otherwise, if only using the hardware technique, the parallel program decomposition should allow starving, that may happen if the program has more threads or processes than the depth L that want to coherently use the same resource.

A non-starving implementation with optional OS support is depicted in Figure 3 and works as decribed below. The critical data stored within the SHECS is considered locked if the lock bit is set for the relevant CAM entry tag. Otherwise if the lock bit is clear, it is considered unlocked and may be reused (it is coherent). If the critical data is not stored within the SHECS it is obviously unlocked. A thread attempting a critical section performs a lookup-lock operation in the explicit CAM system using a data identifier (index or address).

1. CAM miss means that the critical data is unlocked. The SHECS reserves (locks) a least recently used entry and writes its tag with the data identifier, then returns the entry's index within the result miss-entry-reserved, if such an entry is available. If not (all entries are locked within the specified banks and there isn't any entry to be reserved)- the SHECS returns status miss-all-reserved.

    - If the result is miss-entry-reserved the thread must read the critical data from the shared main memory. It should update the CAM entry's cache with the modified data upon exiting from the critical section with CAM-cache-write-unlock operation. After the update,

the critical data resides in the cache. If there were any CAM-lookup-lock operations for the same critical resource in the meantime, they were queued in the lookup-lock FIFO queue in the same CAM bank. In that case the first candidate for entering the critical section is released from the FIFO and signaled to enter the section.

- Otherwise (miss-all-reserved), the thread should be queued by the OS in a software lock-pending queue that also means that it has been swapped out in a software way.

2. CAM hit with detecting lock bit set means that the critical data is locked. In that case the operations are queued in the lookup-lock FIFO queue in the same CAM bank if the FIFO has the available space. In that case the thread is swapped out as long as it will be removed from the FIFO and signaled upon the leaving of critical section by another thread. Otherwise, the result is hit-locked and in that case the OS queues the thread in a software lock-pending queue that also means that it has been swapped out in a software way.

3. CAM hit with detecting lock bit clear state means that the critical data resides in the hit entry's cache and it is unlocked. In that case the lock bit for this CAM entry is automatically set and the critical data with the result hit-unlocked is transported from the cache to the thread registers (feature supported in the network processors) or to the internal cache of the processor executing the thread. After using and modifying, the critical data should be updated in the SHECS's cache and in the main shared memory. The cache is updated with operation CAM-cache-write-unlock that automatically unlocks the CAM entry and consequently if there were any CAM-lookup-lock operations for the same critical resource in the meantime, they were queued in the lookup-lock FIFO queue in the same CAM bank. In that case the first candidate for entering the critical section is released from the FIFO and signaled to enter the section.

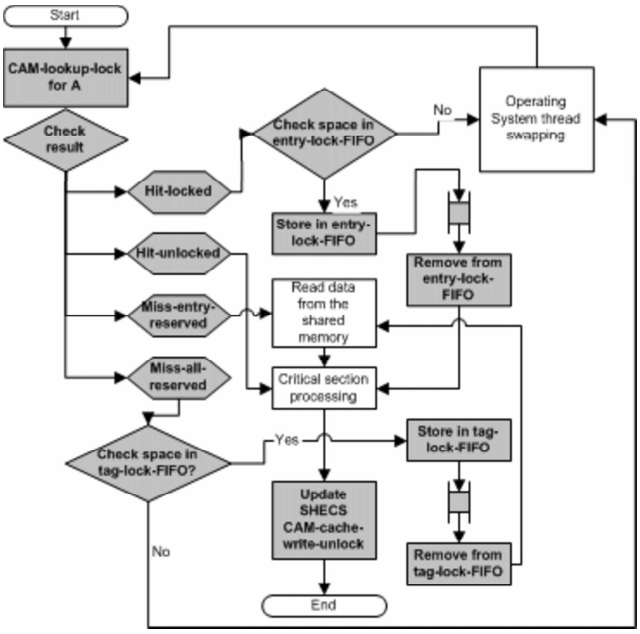

Figure 3. The SHECS's algorithm for full-locking critical sections (gray elements mark hardware processing)

5. SHECS's Simulation on Network Processor

5.1. Simulation method

The SHECS was simulated with the Developer Workbench 4.2 for Intel's IXA Network Processors. Two microengines were used to simulate two SHECS banks with capacity of 32 cacheable entries in total. The latency introduced with the SHECS simulation was assumed comparable with hardware implementation latency. The parallel application was running on four microengines and it implemented the full-locking locking critical section. Every thread in the application performed random data accesses to a table with limited number of entries. The application's algorithm is illustrated with the below pseudo code:

Parallel application pseucode

```
while (no_loops--) {
   random= PSEUDO_RANDOM_NUM mod 32; //or mod 128
   calculate addrA from random;
   cycle_delay(cycle_delay1);
   shecs_lookup_lock(/*in*/addrA, /*out*/entryA, /*out*/indexA);
      modify entryA;
      cycle_delay(cycle_delay2);
   shecs_write_unlock(/*in*/addrA, /*in*/entryA, /*in*/indexA);
}
```

The critical section is between `shecs_lookup_lock()` and `shecs_write_unlock()` primitives. Time spend in the critical section was chosen to be 0.1 of the processing time before it. Thus the critical section was relatively short. The experiments were made with varying the following parameters:

- number of entries in the shared table - 2 values: 32 - equal to the SHECS's capacity, 128 - four times larger than the SHECS's capacity
- number of executing threads on four microengines, values were 32, 16, 8, 4 that were 8, 4, 2, 1 threads per one microengine respectively

5.2. Simulation results

The SHECS possible results were:

- bypassed - critical data had been stored unlocked in the SHECS and after locking it was transfered to a new critical section owner
- locked-bypassed - critical data had been stored locked in the SHECS and after lock release it was transfered to a new critical section owner
- reloaded - critical data had not been stored in the SHECS and the entry's ownership was granterd to a new critical section owner
- locked-reloaded - all SHECS's entries had been used and after freeing one of them its ownership was granted to a new critical section owner

Figure 4 illustrates that the adding of parallel resources might reduce the scalability of speedup, because the number of pending locks increases. Figure 5 and Figure 6 show the distributions of results in cases when the SHECS's capacity is equal and is less to the number of locked shared resources, respectively.

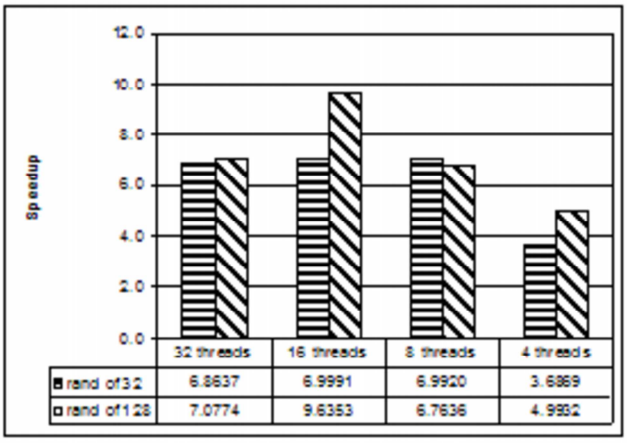

Figure 4. Speedup vs. number of parallel threads and different table sizes

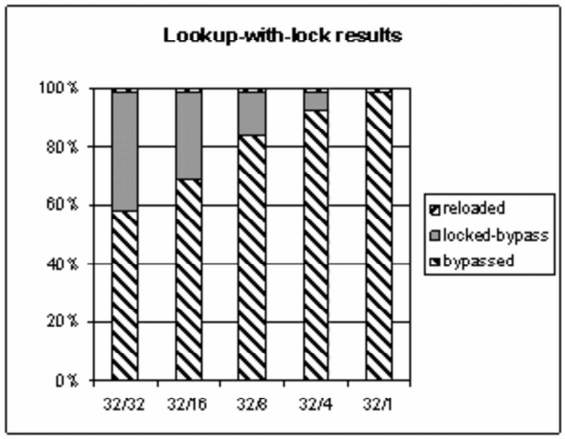

Figure 5. The distribution of results for execution with table size equal to the SHECS's capacity

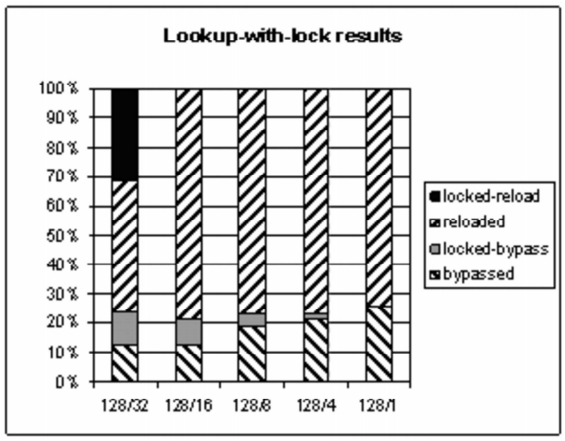

Figure 6. The distribution of results for execution with table size 4 times larger than the SHECS's capacity

6. Final Remarks

The use of SHECS in a parallel system with shared memory architecture should enable achieving the best possible performance gain in the data driven parallelization of sequential programs with required data coherency. The SHECS may be also used to speed-up data searching algorithms, thanks to providing explicit associative searching in CAM banks for a value. Therefore, the parallel algorithms, that search and modify shared dynamic data structures, can benefit from the both SHECS features critical section synchronization with caching data and associative searching. Such features combination constitutes very powerful proposition for the shared memory architectures. The SHECS increases the real parallelism in such systems and also proposes the critical sections support for managing the cache coherence problems. The only disadvantage of the SHECS is the cost - it is an additional, manageable cache system. The content addressable memories (CAMs) (the key ingredient of the SHECS) are still pricy and they aren't used in the general purpose systems. They are still perceived as rather co-processing elements in the network systems in which they are responsible for associative searching of a route for a packet. However the Moore's Law constantly decreases the cost of silicon circuits and it may cause that in the feasible future the SHECS will be implemented and will offer performance gain in the parallel multicore systems integrated on a die.

References

[1] H. Krawczyk, T. Madajczak: Optimal Programming of Critical Sections in Modern Network Processors under Performance Requirements. In the Proc. of IEEE Parelec Conf. Dresden 2004. 2004

[2] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich: The Next Generation of Intel IXP Network Processors. Intel Technology Journal Vol. 6 issue 3. 2002

[3] N. Aboulenein, J. Goodman, S. Gjessing, P. Woest: Hardware support for synchronization in the Scalable Coherent Interface (SCI). In Eighth International Parallel Processing Symposium. 1994

[4] U. Ramachandran, J. Lee: Cache-based synchronization in shared memory multiprocessors. In Supercomputing '95. 1995

[5] M. Adiletta, D. Hooper, M. Wilde, Packet over SONET: Achieving 10 Gbps Packet Processing with an IXP2800. Intel Technology Journal Vol. 6 issue 3. 2002

[6] A. Nanda, A. Nguyen, M. Michael, and D. Joseph: High-Throughput Coherence Controllers. In the Proc. of the 6th Int'l HPC Architecture. 2000

[7] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi: Scalability Port: A Coherent Interface for Shared Memory Multiprocessors. In the Proc. of the 10th Hot Interconnects Symposium. 2002

[8] A. Grbic: Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors. A PhD thesis from Grad. Dep. of Electrical and Computer Engineering University of Toronto. 2003

[9] G. Byrd: Communication mechanisms in shared memory multiprocessors. A PhD thesis from Dep. Of Electrical Engineering of Stanford University. 1998

[10] R. Jin, G. Yang, G. Agrawal: Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming, Interface, and Performance. IEEE Transactions on Knowledge and Data Engineering, Vol. 16, No.10. 2004